WIDS Project Report:
SAT-Based Solvers for Constraint Satisfaction Problems

Majid Husain

Contents

1 Introduction

One of the most fundamental categories of problems in the fields of computer science and artificial intelligence is known as constraint satisfaction problems (CSPs). An example of a CSP is a collection of variables, domains for each variable, and a number of constraints that limit the types of assignments that are acceptable. Discovering a task that simultaneously satisfies all of the constraints is the objective of this endeavor.

This project investigates Boolean Satisfiability (SAT) as a general CSP resolution framework. Instead of creating a new algorithm for each problem, SAT-based approaches encode the problem in propositional logic and use a SAT solver to determine satisfiability.

This project aimed to gain an internal understanding of SAT solvers instead of viewing them as mysterious. Two representative problems, Sudoku and Sokoban, were used to test classical algorithms like DPLL and CDCL, which were created from scratch.

2 Theoretical Background

Finding a satisfying assignment for a Boolean formula is known as the SAT problem. Since SAT is NP-complete, its worst-case time complexity must be exponential.

Modern SAT solvers, however, work incredibly well in real-world scenarios. The reason for this is that they don't try to list every assignment. Instead, they significantly lower the average-case complexity by employing logical reasoning strategies like unit propagation, clause learning, and non-chronological backtracking.

The goal of this project is to comprehend these methods through practical applications.

3 Worst-Case vs Average-Case Complexity

There are 2^n possible assignments for a CNF formula with n Boolean variables. In the worst scenario, this entire space might need to be explored by a SAT solver. This worst-case complexity remains unchanged even for advanced solvers based on CDCL.

The ability of SAT solvers to lower the average-case complexity is what makes them successful. Unit propagation quickly forces assignments, clause learning prevents repeating the same mistakes, and backjumping skips irrelevant parts of the search tree.

Thus, SAT solvers do not the eliminate exponential complexity problem but they provide the best case optimization.

4 Basic DPLL Algorithm

The DPLL algorithm is a method for solving SAT that involves recursive backtracking. It alternates between forced assignments (unit propagation) and decision assignments (branching).

4.1 DPLL Code

```
def dpll(cnf, assignment={}):
    def simplify(cnf, var, val):
        new=[]
        for clause in cnf:
            if (var if val else "~" + var) in clause:
                continue
            new_clause=[l for l in clause
                        if l!=("~" + var if val else var)]
            new.append(new_clause)
        return new

    for clause in cnf:
        if len(clause) == 1:
            lit = clause[0]
            var = lit.strip("~")
            val = not lit.startswith("~")
            return dpll(
                simplify(cnf, var, val),
                {**assignment, var: val}
            )

    if cnf==[]:
        return True, assignment

    if [] in cnf:
```

```
        return False, None

    lit = cnf[0][0]
    var = lit.strip("~")

    sat, sol = dpll(
        simplify(cnf, var, True),
        {**assignment, var: True}
    )
    if sat:
        return True, sol

    return dpll(
        simplify(cnf, var, False),
        {**assignment, var: False}
    )
```

4.2 Explanation

The classical DPLL framework is used with the implementation. The first step is to handle unit clauses - this forces variables to be assigned a specific value. If there are no unit clauses, a variable is selected and both true and false value possibilities are tried recursively.

The algorithm will find the correct answer; however, it does not learn from conflicts, so it is inefficient.

5 Conflict-Driven Clause Learning (CDCL)

To improve optimization, clause learning is introduced to the algorithm. CDCL extends DPLL by recording conflicts and 'learning them' to reduce future search.

5.1 Basic CDCL Code

```
def cdcl(cnf, assignment=None, level=0, trail=None):
    if assignment is None:
        assignment = {}
```

```
    if trail is None:
        trail = []

    def simplify(cnf, var, val):
        new=[]
        for clause in cnf:
            if (var if val else '~'+var) in clause:
                continue
            new_clause=[lit for lit in clause
                        if lit!=('~'+var if val else var)]
            new.append(new_clause)
        return new

    changed=True
    while changed:
        changed=False
        for clause in cnf:
            if len(clause)==1:
                lit=clause[0]
                var=lit.strip("~")
                val=not lit.startswith("~")
                assignment[var]=val
                trail.append((var,val,level))
                cnf=simplify(cnf,var,val)
                changed=True
                break

    if cnf==[]:
        return True, assignment

    if [] in cnf:
        return False, None

    lit=cnf[0][0]
    var=lit.strip("~")

    sat,res=cdcl(
        simplify(cnf,var,True),
```

```
        {**assignment,var:True},
        level+1,
        trail+[(var,True,level+1)]
    )

    if sat:
        return True,res

    learned=[]
    for v,val,lvl in trail:
        if lvl==level+1:
            learned.append('~'+v if val else v)

    if learned:
        cnf.append(learned)

    return cdcl(
        simplify(cnf,var,False),
        {**assignment,var:False},
        level+1,
        trail+[(var,False,level+1)]
    )
```

5.2 Explanation

This version introduces clause learning. When a conflict occurs, a new clause is added to prevent the same conflict from being repeated. This improves the performance by a lot already.

6 Optimized CDCL with Backjumping

A more structured CDCL solver introduces decision levels and non-chronological backtracking. This is a highly optimized and complex CDCL algotrithm

6.1 Optimized CDCL Code

```
def cdcl(cnf):
    assignment={}
    decision_level={}
    trail=[]
    level=0

    def var_of(lit):
        return lit[1:] if lit.startswith("~") else lit

    def val_of(lit):
        return not lit.startswith("~")

    def negate(lit):
        return lit[1:] if lit.startswith("~") else "~"+lit

    def unit_propagate():
        for clause in cnf:
            unassigned=[]
            satisfied=False
            for lit in clause:
                v=var_of(lit)
                if v in assignment:
                    if assignment[v]==val_of(lit):
                        satisfied=True
                        break
                else:
                    unassigned.append(lit)
            if satisfied:
                continue
            if len(unassigned)==0:
                return clause
            if len(unassigned)==1:
                v=var_of(unassigned[0])
                assignment[v]=val_of(unassigned[0])
                decision_level[v]=level
                trail.append(v)
        return None
```

6.2 Explanation

This solver explicitly tracks decision levels and allows backjumping to the level responsible for a conflict. This closely resembles industrial SAT solvers.

7 Sudoku: Problem and SAT Encoding

Sudoku is a static logic puzzle defined on a 9×9 grid. The goal is to assign digits from 1 to 9 such that each digit appears exactly once in every row, column, and 3×3 subgrid.

Sudoku is naturally a CSP (constraint satisfying problem). There is no notion of time or action; the task is to find a single consistent assignment.

7.1 Boolean Representation

Boolean variables $X_{i,j,k}$ represent whether digit k is placed in cell (i, j). This converts Sudoku into a purely Boolean problem with 729 variables.

7.2 Constraint Encoding

Constraints enforce:

- Exactly one digit per cell
- Unique digits per row
- Unique digits per column
- Unique digits per subgrid

Pre-filled cells are encoded as unit clauses (to remain consistent with the initial assignment).

7.3 How the Solver Solves Sudoku

The SAT solver reasons logically:

- Unit propagation fixes forced digits
- Branching corresponds to guessing a digit
- Conflicts arise from violated constraints
- Clause learning prevents repeating mistakes

Thus, Sudoku is solved by systematic logical elimination rather than brute-force guessing.

8 Sokoban: Game Description and SAT Modeling

Sokoban is a Japanese puzzle game in which a player moves inside a grid and pushes boxes onto target locations. Boxes cannot be pulled, and incorrect moves may lead to irreversible deadlocks.

Unlike Sudoku, Sokoban is a dynamic planning problem. A solution is a sequence of actions over time.

8.1 Why Sokoban Is Hard

Sokoban has a large state space, irreversible moves, and long dependency chains.

8.2 SAT Encoding of Sokoban

Sokoban is encoded by unrolling time. Boolean variables represent player and box positions at each time step, as well as actions.

Constraints enforce valid movement, conservation of boxes, and goal satisfaction at the final time step.

8.3 SAT Reasoning vs BFS

Unlike BFS, SAT reasons globally across all time steps. Clause learning eliminates entire classes of failing paths, making SAT effective for constrained instances.

9 Conclusion

This project demonstrated how SAT solvers can be used to solve both static and dynamic constraint satisfaction problems. By implementing DPLL and CDCL from scratch and applying them to Sudoku and Sokoban, the project provided deep insight into logical reasoning, complexity, and solver behavior.